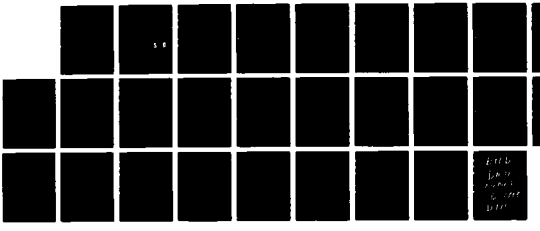
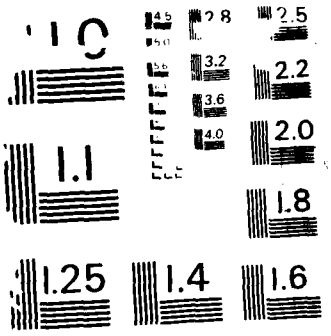
AD-A192 321 PARALLEL SOLUTIONS TO GEOMETRIC PROBLEMS ON THE SCAN MODEL OF COMPUTATION.. (U) MASSACHUSETTS INST OF TECH CAMBRIDGE ARTIFICIAL INTELLIGENCE L.. 1/1

UNCLASSIFIED G E BLELLOCH ET AL. FEB 88 AI-M-952 F/G 12/2 NL





TEST CHART

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

AD-A192 321

REPORT DOCUMENTATION PAGE

1 REPORT NUMBER	2 GOVT ACCESSION NO	3 RECIPIENT'S CATALOG NUMBER
AIM-952		

4 TITLE (and Subtitle)

Parallel Solutions to Geometric Problems on the Scan Model of Computation

5 TYPE OF REPORT & PERIOD COVERED

AI-Memo

6 PERFORMING ORG. REPORT NUMBER

7 AUTHOR(s)

Guy E. Blelloch and
James J. Little

8 CONTRACT OR GRANT NUMBER(s)

DACA76-85-C-0010
N00014-85-K-0124

9 PERFORMING ORGANIZATION NAME AND ADDRESS

Artificial Intelligence Laboratory
545 Technology Square
Cambridge, MA 02139

10 PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS

11 CONTROLLING OFFICE NAME AND ADDRESS

Advanced Research Projects Agency
1400 Wilson Blvd.
Arlington, VA 22209

12 REPORT DATE

February 1988

13 NUMBER OF PAGES

32

14 MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office)

Office of Naval Research
Information Systems
Arlington, VA 22217

15 SECURITY CLASS. (of this report)

UNCLASSIFIED

15a DECLASSIFICATION/DOWNGRADING SCHEDULE

16 DISTRIBUTION STATEMENT (of this Report)

Distribution is unlimited.

17 DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18 SUPPLEMENTARY NOTES

None

19 KEY WORDS (Continue on reverse side if necessary and identify by block number)

Computational Geometry, Parallel Computation, Connection Machine, Graphics

20 ABSTRACT (Continue on reverse side if necessary and identify by block number)

This paper describes several parallel algorithms that solve geometric problems. The algorithms are based on a vector model of computation - the scan-model. The purpose of this paper is both to show how the model can be used and to show a set of interesting algorithms.

We describe a k-D tree algorithm that, for n points, requires O(lg n) calls to the primitives, a closest-pair algorithm that requires O(lg n) calls to the primitives, a line-drawing algorithm that requires O(1) calls to the primitives, a line-of-sight algorithm that requires O(1) calls to the

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE
S/N 0102-014-6601

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

Block 20 · cont.

primitives, and finally three different convex-hull algorithms. All these algorithms should be noted for their simplicity rather than complexity; many of them are parallel versions of known serial algorithms.

Most of the algorithms discussed in this paper have been implemented on the Connection Machine, a highly parallel single instruction multiple data (SIMD) computer.

MASSACHUSETTS INSTITUTE OF TECHNOLOGY
ARTIFICIAL INTELLIGENCE LABORATORY

A.I. Memo 952

February, 1988

Parallel Solutions to Geometric Problems on the Scan Model of Computation

Guy E. Blelloch and James J. Little

Abstract

This paper describes several parallel algorithms that solve geometric problems. The algorithms are based on a vector model of computation — the *scan-model*. The purpose of this paper is both to show how the model can be used and to show a set of interesting algorithms.

We describe a k-D tree algorithm that, for n points, requires $O(\lg n)$ calls to the primitives, a closest-pair algorithm that requires $O(\lg n)$ calls to the primitives, a line-drawing algorithm that requires $O(1)$ calls to the primitives, a line-of-sight algorithm that requires $O(1)$ calls to the primitives, and finally three different convex-hull algorithms. All these algorithms should be noted for their simplicity rather than complexity; many of them are parallel versions of known serial algorithms.

Most of the algorithms discussed in this paper have been implemented on the Connection Machine, a highly parallel single instruction multiple data (SIMD) computer.

Acknowledgements: This report describes research done within the Artificial Intelligence Laboratory at the Massachusetts Institute of Technology. Support for the A.I. Laboratory's artificial intelligence research is provided in part by the Advanced Research Projects Agency of the Department of Defense under Army contract DACA76-85-C-0010 and in part under the Office of Naval Research contract N00014-85-K-0124.

1 Introduction

The purpose of this paper is twofold. Firstly, it describes a set of elegant, practical algorithms for solving a diverse set of problems in computational geometry and graphics. Secondly, it helps demonstrate that the *scan-model* is a viable model of computation. These two purposes complement each other: the model allows a simple description of the algorithms, and the algorithms demonstrate the power of the model.

Researchers have suggested several synchronous parallel models of computation. The most popular of these models are the parallel random access machine (P-RAM) models [13]. A P-RAM consists of a set of conventional processors attached to a single shared memory. Processors communicate through the shared memory: one processor can write a value into the memory and another processor can read this value. Researchers have suggested several variations of the P-RAM models. These variations mostly differ in whether or not they permit concurrent reads from, or concurrent writes to, a unique memory location. By assuming that memory references take *unit-time*, the P-RAM models have been used to determine the asymptotic running time of many parallel algorithms.

We suggest another class of synchronous parallel models of computation defined in terms of a set of primitive operations that work on arbitrarily long vectors of simple values. We call these models, vector models. The models differ from P-RAM models both in that they are single instruction multiple data (SIMD) models, and in that there is no concept of a memory shared among many processors. Elements in a vector communicate through a permutation primitive rather than a shared memory. As with the P-RAM models, vector models can be used to analyze the asymptotic running time of algorithms, by assuming that a set of primitives take *unit-time*.

Since vector models are SIMD, they can be efficiently mapped onto a wider range of architectures than P-RAM models can. As well as being implementable on standard serial computers and on multiple instruction parallel computers, they can be efficiently implemented on vector processors, such as the vector processor of the CRAY systems [24], or single instruction parallel computers, such as the Connection Machine [16]. On the other hand, since P-RAM models are multiple instruction multiple data (MIMD) models, they are more powerful than vector models. As should become evident in this paper, and as shown elsewhere [10], this additional power is not necessary for a broad range of practical algorithms. We also believe that vector models tend to lead to simpler and more concrete algorithm descriptions than do P-RAM models.

The *scan-model* is a particular vector model in which three classes of vector operations are considered *unit-time* primitives: elementwise arithmetic and logical operations;

permutation operations: and scan operations, a type of prefix computation. By *unit-time* primitives we mean that they require approximately an equivalent duration of time when executed on equal length vectors.

In this paper we describe several algorithms based on the scan-model. The first is an algorithm that constructs a k-D tree. A k-D tree is a technique for splitting n points in a k dimensional space into n regions each with a single point. The k-D tree technique is used as a substep in a large number of applications ranging from rendering images to machine learning [20]. For n points, the algorithm we describe takes $O(k \lg n)$ calls to the primitives using vectors of length $O(n)$. This algorithm is optimal in the sense that even if simulated on a serial machine, it will run in the same asymptotic running time as the best serial algorithm.

Based on the k-D tree algorithm, we describe a two dimensional closest-pair algorithm. In the two dimensional closest pair problem we want to find the pair of points in a plane that are closest to each other (Euclidean distance). This algorithm is a parallel version of an algorithm of Bentley and Shamos [9]. For n points in a two dimensional space, our algorithm requires $O(\lg n)$ calls to the primitives using vectors of length $O(n)$.

The third algorithm is a line drawing routine. Line drawing is the problem of: given a pair of points on a two dimensional grid (the two endpoints of a line), determine what *pixels in a finite resolution grid lie on* a line between the endpoints. This routine requires $O(1)$ calls to the primitives using vectors no longer than the number of points in the line. The routine has been extended to render solid objects [25].

The fourth algorithm is a line of sight algorithm. Given a grid of altitudes and an observation point on the grid, the algorithm returns the points visible from the observation point. A line of sight algorithm can be applied to help determine where to locate potential eyesores. For example, when designing a building, a highway or a city dump, it is often informative to know from where the "eyesore" will be visible.

We finally describe three planar convex-hull algorithms. Given n points in the plane, the planar convex hull problem finds which of these points lie on the perimeter of the smallest convex region that contains all points. Two of the convex-hull algorithms we describe are simple and are likely to perform very well in practice, but they are not provably optimal — certain sets of carefully selected points will perform badly. The third algorithm is more complicated and probably less practical, but is theoretically optimal. This algorithm is based on a parallel algorithm designed for the concurrent read exclusive write (CREW) P-RAM model [14].

Most of the algorithms we describe in this paper have been implemented on the Con-

nection Machine. The code we show in the text with some syntactic changes is actual code used to execute the algorithms.

The remainder of this paper is organized as follows:

- We define the scan-model in terms of the primitive operations it supports.
- We introduce some powerful techniques based on the scan-model. These techniques are used extensively in the description of algorithms.
- We describe the algorithms.

2 The Scan-Model

The scan-model is defined in terms of a set of primitive operations that operate on arbitrarily long vectors of atomic values. By a vector we mean a one dimensional array (an ordered set). By atomic values we mean values that can be represented in $O(\lg n)$ bits — in this paper we only use integers, floating point numbers and boolean values. We assume that all primitives require approximately an equivalent duration of time when operating on equal length vectors. We call this time "unit time". To determine the actual running time of an algorithm on a particular machine, we need to know both the number of calls to the primitives and the length of the vectors used[1].

The scan-model has three classes of unit-time primitives: elementwise arithmetic and logical operations, permutation operations, and scan operations, a type of parallel prefix computation.

Elementwise Primitives

Each elementwise primitive operates on equal length vectors, producing a result vector of equal length. The element i of the result is an elementary arithmetic or logical primitive — such as $+, -, *$, or and not — applied to element i of each of the input vectors. For example:

[1] The vector length is important even on parallel machines since for sufficiently long vectors, multiple elements must be allocated to each processor and each processor must loop over these elements when executing an operation.

A	=	[5	1	3	4	3	9	2	6]
B	=	[2	5	3	8	1	3	6	2]
A + B	=	[7	6	6	12	4	12	8	8]
A × B	=	[10	5	9	24	3	27	12	12]

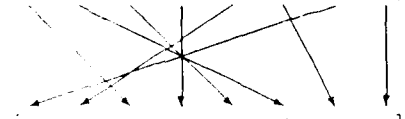
In addition to the standard elementary operations, we include an operator select that takes one boolean argument and two other arguments. Based on the boolean argument, the select function will return either the first or second of the other two arguments.

A	=	[5	1	3	4	3	9	2	6]
B	=	[2	5	3	8	1	3	6	2]
F	=	[T	F	F	F	T	T	F	T]
select(F, A, B)	=	[5	5	3	8	3	9	6	6]

Permutation Primitives

The *permutation* primitive takes two vector arguments -- a *data vector* and an *index vector* and permutes each element in the data vector to the location specified in the index vector. For example:

Vector Index	=	[0	1	2	3	4	5	6	7]
A (data vector)	=	[o	t	e	m	e	r	g	y]
I (index vector)	=	[2	5	4	3	1	6	0	7]
permute(A, I)	=	[g	e	o	m	e	t	r	y]



It is an error for more than one element to have the same index -- the permutation must be one-to-one. This restriction is similar to the restriction made in the exclusive read exclusive write (EREW) P-RAM model, in which it is an error to write more than one value to a particular memory location at a time.

To allow communication between vectors of different sizes, we include a version of the permute primitive that returns a vector of different length than the source vectors. This version takes two extra arguments: a *default vector*, which specifies the length of the destination vector and puts default values in positions that do not receive any value; and a *selection vector*, which masks out certain elements so that they do not permute. For example:

Vector Index	=	[0	1	2	3	4	5	6	7]
A (data vector)	=	[o	t	e	m	e	r	y	s]
D (default vector)	=	[f	r	e	e]				
S (selection vector)	=	[T	F	F	T	F	F	F	F]
I (index vector)	=	[2	5	4	3	1	6	7	0]
permute(A, I, S, D)	=	[f	r	o	m]				

Scan Primitives

The scan primitives execute a scan operation, sometimes called a *prefix computation*, on a vector. The scan operation takes a binary associative operator $\oplus$, and a vector $[a_0, a_1, ..., a_{n-1}]$ of n elements, and returns the vector $[a_0, (a_0 \oplus a_1), ..., (a_0 \oplus a_1 \oplus ... \oplus a_{n-1})]$. In this paper we will only use plus, maximum, minimum, or and and as operators for the *scan primitives*. We will henceforth call these scan operations +-scan, max-scan, min-scan, or-scan and and-scan. Some examples:

A	=	[5	1	3	4	3	9	2	6]
+-scan(A)	=	[5	6	9	13	16	25	27	33]
max-scan(A)	=	[5	5	5	5	5	9	9	9]

Some readers might be skeptical about considering the scan operations as "unit-time" primitives. Our justification is straightforward. On a serial machine, it is clear that the scan operations using simple operators such as + will be just as fast as the other primitives: all the primitives will take $O(n)$ time on vectors of length n. On a parallel machine it is not hard to show, both in theory and in practice, that a circuit that executes the scan operations can be built with less hardware and will run just as fast, or faster, than a circuit that executes a read or write into a shared memory (such a read or write can be used to implement the permutation primitive). This is argued in more detail in [11]. Admittedly, both the scan and a shared memory reference take at least $\lg n$ real time, but we are only arguing here that the primitives take approximately the same amount of time on equal length vectors.

In the description of algorithms we will often loosely refer to vectors in which each element contain more than one atomic value. For example, we will use vectors of points in a two dimensional space; each point has two values, an x and y coordinate, so the vector

$$[(3, 6) \quad (4, 5) \quad (9, 7)]$$

represents the three points (3, 6), (4, 5) and (9, 7). At the primitive such a structure vector would be implemented with two vectors but a higher level language could support record-like vectors in which each element has some constant number of values.

2.1 Segments

This section describes a method that allows a programmer to take a vector *routine*[2] defined to work on a single set of data and then apply it to many sets in parallel. For example, if we had a vector routine that sorted a set of values, we could apply it to sort many sets of data in parallel. Or, if we had a vector routine that, given endpoints, determines the pixels on a line, we could apply it to draw many lines in parallel.

The technique involves dividing a vector into segments and placing one set of data in each segment. To keep track of how a data vector is segmented, we associate with the data vector a *segment-descriptor*. A *segment-descriptor* is itself a vector which has as many elements as segments of the data vector; each of these elements contains an integer which specifies the length of the segment[3]. For example:

A'	=	[5	1	3	4	3	9	2	6]
segment-descriptor	=	[2	4	2]					
A	=	[5	1]	[3	4	3	9]	[2	6]

Henceforth, the notation

$$A = [5 \quad 1] \quad [3 \quad 4 \quad 3 \quad 9] \quad [2 \quad 6]$$

is shorthand for a pair of vectors: the data vector along with its *segment-descriptor*.

For each primitive of the scan-model we define a segmented version that works independently within each segment. Figure 1 shows examples of segmented versions of the primitives. The segmented version of the permutation primitive bases its indices relative to the beginning of each segment so values permute *within* a segment — it is an error for an index to reference outside of the segment. The segmented version of the scans primitives restart at the beginning of each segment[4]. The segmented version of the elementwise operations are unchanged.

---

[2] A vector routine is a routine defined in terms of the vector primitives we discussed.

[3] There are several other ways of representing segments [10] but we find this representation the most convenient.

[4] A similar operation was suggested by Schwartz [20].

A	=	[5	1]	[3	4	3	9]	[2	6]
B	=	[1	0]	[2	0	3	1]	[0	1]
+-scan(A)	=	[5	6]	[3	7	10	19]	[2	8]
max-scan(A)	=	[5	5]	[3	4	4	9]	[2	6]
permute(A, B)	=	[1	5]	[4	9	3	3]	[2	6]

Figure 1: Examples of the segmented versions of the primitive operations.

All the segmented versions can be simulated with a small constant number of calls to the unsegmented versions [10], but they are so useful that in practice they might be implemented directly. We will henceforth assume that the segmented versions of the primitives are themselves primitives.

We now return to the initial claim of this section:

The Segment Lemma: With a segmented version of all the primitives of the scan-model, we can apply any routine defined in terms of those primitives to work on a single set of data, to multiple sets of data independently and in parallel.

We won't prove this lemma in this paper, but it should be intuitive; a proof is given in [10]. This lemma allows great simplification of the code needed to describe parallel algorithms.

3 Some Simple Operations

In this section we describe several useful, simple operations that can be implemented with a small constant number of calls to the primitive operations [11]. As with the segmented versions of the primitives, these operations are useful enough that they might themselves be considered primitives and be implemented directly.

distribute *values lengths*

The distribute operation takes a vector of *values* and a vector of *lengths* and distributes each value into a segment of length specified by *lengths*. For example:

A	=	[7	3	8]					
L	=	[2	4	2]					
distribute(A, L)	=	[7	7]	[3	3	3	3]	[8	8]

A similar operation was first suggested by Batcher [6] — he called it an irregular spreading.

index *lengths*

The index operation takes a vector of *lengths*, creates a segment for each length, and returns the index of each element within each segment. For example:

L	=	[2 4 2]
index(L)	=	[0 1] [0 1 2 3] [0 1]

element *values indices*

The element operation takes a segmented vector *values*, and a vector of *indices* with one element per segment. Each index i is used to extract the i^{th} element from the corresponding segment in *values*. For example:

A	=	[5 1] [3 4 3 9] [2 6]
I	=	[0 2 1]
element(A, I)	=	[5 3 6]

+-reduce *values*

The *reduce* operations takes a segmented vector of *values* and combines all the elements in each segment using one of five binary operators: +, maximum, minimum, or or and. It returns a vector with as many elements as segments.

Some Examples:

A	=	[5 1] [3 4 3 9] [2 6]
+-reduce(A)	=	[6 19 8]
max-reduce(A)	=	[5 9 6]

append *values1 values2*

The append operation takes two segmented vectors of *values* with the same number of segments. And appends the two vectors segmentwise. For example:

A	=	$[a_{00}\ a_{01}\ a_{02}]$ $[a_{10}]$ $[a_{20}\ a_{21}]$
B	=	$[b_{00}]$ $[b_{10}\ b_{11}]$ $[b_{20}\ b_{21}]$
append(A, B)	=	$[a_{00}\ a_{01}\ a_{02}\ b_{00}]$ $[a_{10}\ b_{10}\ b_{11}]$ $[a_{20}\ a_{21}\ b_{20}\ b_{21}]$

pack *values flags*

The pack operation takes a segmented vector of *values* and a segmented boolean vector of *flags*, and packs all the elements with a T in their flag into consecutive elements, deleting elements with an F in their flag. For example:

A	=	[5	1]	[3	4	3	9]	[2	6]
F	=	[T	F]	[T	F	F	T]	[T	T]
pack(A, F)	=	[5]		[3	9]			[2	6]

A similar operation was first suggested by Batcher [1] he called it an irregular compression.

split *values flags*

The split operation takes a segmented vector of *values* and a segmented boolean vector of *flags*, and packs all the elements with an F in their flag to the bottom of each segment and elements with a T in their flag to the top of each segment. It also splits each segment in two at the boundary between the T and F elements. For example:

A	=	[5	1]	[3	4	3	9]	[2	6]
F	=	[T	F]	[T	F	F	T]	[T	T]
split(A, F)	=	[1]	[5]	[4	3]	[3	9]	[]	[2	6]

We also define a delete-split operation which is the same as split but deletes any empty segment.

A	=	[5	1]	[3	4	3	9]	[2	6]
F	=	[T	F]	[T	F	F	T]	[T	T]
delete-split(A, F)	=	[1]	[5]	[4	3]	[3	9]	[2	6]

rank-split *ranks flags*

The rank-split operation is similar to the split operation except that the *ranks* argument must be a valid set of indices for the permutation primitive. As well as splitting these indices, the rank-split operation renumbers them so they are valid within the new segments but maintain the same order. For example:

A	=	[1	0]	[2	1	3	0]	[0	1]
F	=	[T	F]	[T	F	F	T]	[T	T]
rank-split(A, F)	=	[0	0]	[0	1]	[1	0]	[]	[0	1]

Key	=	[4	7	2	1	5	3	7	2]
Pivot	=	5							
Key ≥ Pivot	=	[F	T	F	F	T	F	T	F]
delete-split	=	[4	2	1	3	2]	[7	5	7]
Pivot Value	=	3					7		
Key ≥ Pivot	=	[T	F	F	T	F]	[T	F	T]
delete-split	=	[2	1	2]	[4	3]	[5]	[7	7]
Pivot Value	=	2			4		5	7	
Key ≥ Pivot	=	[T	F	T]	[T	F]	[T]	[T	T]
delete-split	=	[1]	[2	2]	[3]	[4]	[5]	[7	7]

Figure 2: An example of parallel Quicksort. Each pivot is picked at random from within a segment.

In this example, the F part of the second segment starts with the indices 1 and 3; these are renumbered to 0 and 1 so that they represent a valid index set for the new segments and maintain the same order. The rank-split operation is used to update pointers when performing a split operation.

3.1 Recursive Splitting

The segment abstraction and the primitives we described allow simple definitions of recursive routines that start with some set of values, split this set into subsets and recursively solve the problem on each subset. We will call this technique *recursive splitting*. As an example of such a technique, consider the following parallel version of Quicksort. As with the serial algorithm, the algorithm picks one of the keys as a pivot value, splits the keys into two sets, one with greater valued keys and one with lesser valued keys, and then recurses on each set.

Figure 2 shows an example of the parallel version. The routine picks a random element from each segment as a pivot value using the element operation[5]. The algorithm distributes this pivot value over each segment using a distribute operation, and splits the keys based on whether a key is greater or less than the pivot using the delete-split operation[6]. The

[5] I assume that there is a primitive elementwise random operation which in each element takes an integer and returns a pseudo-random number less than that integer.

[6] We use the delete-split operations instead of the split operation so that we never have more segments

algorithm is now applied recursively to the result. When the numbers within all segment are in non decreasing order, we return and merge the split sets. As with the serial algorithm, this algorithm is expected to complete in $O(\lg n)$ steps[7]. In the scan-model, each step requires a small constant number of operations.

The code needed to implement quicksort in the scan-model is as follows:

```
define quicksort(keys){
    if-any (shift-left(keys) < keys)
      then pivots ← element(keys, random(length(keys)))
           quicksort(delete-split(keys, (distribute(pivots, length(keys)) ≤ keys)));
      else keys}
```

This general recursive splitting technique can be used in most divide and conquer algorithms. In this paper we will use it in the k-D tree algorithm discussed in Section 4, the quickhull algorithm discussed in Section 8.1, and the binary tree search method discussed in Section 9.

3.2 Allocation

Another useful technique is allocation. Many problems require the allocation of a set of elements that can then be operated on in parallel. For example consider a line drawing algorithm that takes as input two endpoints, calculates the length in pixels of the line, and then allocates an element for each pixel so that it can calculate the pixel positions in parallel (this is an outline of the algorithm we discuss in Section 6). Also assume that several lines need to be drawn in parallel.

Such allocation is trivial with the operations we defined in Section 3. If we have an integer vector, in which each element specifies how many new positions it needs, we can use this vector directly in the distribute and index operations to distribute the elements to appropriately sized segments. Such allocation is used in the line drawing routine described in Section 6 and in the line of sight algorithm described in Section 7.

---

than elements.

[7] This is actually only true if either the keys are unique, or we split into three groups at each step ($<$, $=$, $>$), or we switch between $\leq$ and $>$ on alternating steps.

4 Building a k-D Tree

A k-D tree is technique for splitting n points in a k dimensional space into n regions each with a single point [8]. It starts by splitting the space in two along one of the dimensions using a $k-1$ dimensional plane. It then recursively splits each of the subspaces in two. Figure 3 shows an example of a 2-D tree. At each step the algorithm must select which dimension to split within each subspace; the criterion for selection depends on how the tree will be used. A common criterion is to select the dimension along which the spread of points is greatest.

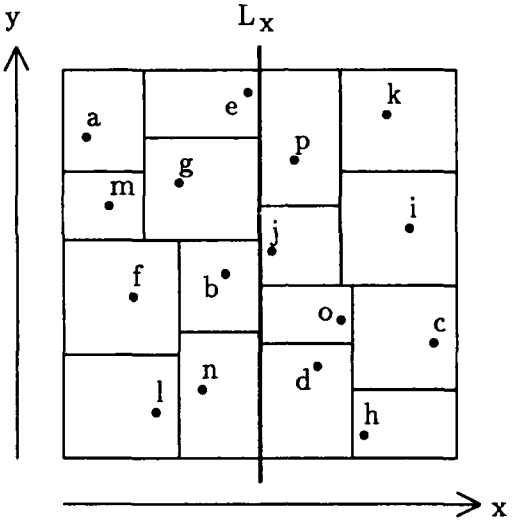
The k-D tree is often used as a step in other algorithms. 3-D trees are used in ray tracing algorithms for rendering solid objects. In such algorithms, objects need only be stored in the regions they penetrate and rays need only examine regions they cross. This can greatly reduce the number of objects each ray needs to examine. k-D trees are also used in many proximity algorithms such as the all closest pairs problem [15] or the closest pair problem, discussed in next section. k-D trees have also been suggested for use in some machine learning algorithms [20].

The algorithm we describe here is a parallel version of a standard serial algorithm [22]. For n points, our algorithm takes $O(k \lg n)$ calls to the primitives on vectors of length n. This algorithm is optimal in the sense that even if simulated on a serial machine, it will run in the same asymptotic running time as the best serial algorithm.

In many k-D tree algorithms, when splitting a space, one point is selected as the split point, and this point in placed in neither side -- it is used to divide the two sides. In our algorithm, when splitting a space, we place all points in one of the two sides — we assume the split line lies half way between the points on either side of the split. For this reason, the algorithm might be more appropriately called a k-D splitting rather than a k-D tree.

Our algorithm consists of one step per split. Each step requires $O(k)$ calls to the primitives. Before executing any steps, the algorithm sorts the set of points according to each of the k dimensions. The sorting can be executed with the Quicksort algorithm discussed earlier, an enumerate-pack sorting algorithm discussed in [11], or a version of Cole's sorting algorithm [12]. Instead of keeping the actual values in sorted order for each dimension, we keep the rank of each point along each dimension. The rank of a point is the position the point would be located at if the vector were sorted. We call the vectors that hold these ranks, *rank-vectors* — there is one *rank-vector* for each dimension. Figure 3 shows an example for a 2-D tree, the initial *rank-vectors*, and the result of the first step.

At each step of the algorithm the *rank-vectors* will contain a segment for each subspace,



point	=	[a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p]
x-rank	=	[0	6	15	10	7	2	4	12	14	8	13	3	1	5	11	9]
y-rank	=	[13	7	4	3	15	6	11	0	9	8	14	1	10	2	5	12]
above-split-line?		[F	F	T	T	F	F	F	T	T	T	T	F	F	F	T	T]
rank-split x-rank	=	[0	6	7	2	4	3	1	5]	[7	2	4	6	0	5	3	1]
rank-split y-rank	=	[6	3	7	2	5	0	4	1]	[2	1	0	5	4	7	3	6]

Figure 3: An example of a 2-D tree. The top diagram shows the final splitting. The vectors below are generated during the first step — when splitting along the line L_x.

and the ranks within each segment will be the correct ranks for that subspace. It suffices to demonstrate that we can execute a split along any dimension and generate new ranks within the two subspaces. The algorithm is then correct by induction.

To split along a given dimension the algorithm distributes the cut line and determines for each point whether it is above or below the line[8]. The algorithm now uses the rank-split operation defined in Section 3 to split each *rank-vector* based on whether a point is below or above the split line. The rank-split operation as defined correctly generates the rank within each subspace. Each step therefore requires $O(k)$ calls to the primitives: some operations to determine whether each point is below or above the split, and k rank-split operations. Since there are $O(\lg n)$ steps, the whole algorithm requires $O(k \lg n)$ calls to the primitives.

In the closest-pair algorithm discussed next it is useful to keep the rank-vectors for all the steps. This will require that we store $k \lg n$ vectors of length n.

5 Closest Pair

In a two dimensional closest pair problem we want to find the pair of points in a plane that are closest to each other (Euclidean distance). The algorithm we describe is a parallel version of an algorithm described by Bentley and Shamos in [9]. For n points, it requires $O(\lg n)$ calls to the primitives using vectors of length $O(n)$. This algorithm requires $O(n \lg n)$ memory ($O(\lg n)$ vectors of length $O(n)$) but can be modified to run with $O(\lg n \lg \lg n)$ calls to the primitives using $O(n)$ memory. Atallah and Goodrich have shown an $O(\lg n \lg \lg n)$ time $O(n)$ processor algorithm to solve the closest pair problem in the concurrent read exclusive write (CREW) P-RAM model.

Our algorithm consists of building the 2-D tree as defined in the previous section[9], and then merging rectangles back to the original region. Given two adjacent rectangles and their closest pairs, a merge step can determine the closest pair of the merged rectangle with a constant number of calls to the primitives. Because of segments, we can merge many pairs of rectangles in parallel.

Since we have already defined how to build the 2-D splitting, we will only describe the *merging* phase. The merging works on the same principle as described in [9]. We will first review the principle and then show how it is implemented on the scan-model. We will denote the separation of the closest pair in a rectangle R by δ_R.

[8] As stated earlier, the method for choosing a cut line will depend on the particular use of the k-D tree.

[9] In this algorithm it does not matter in what order we pick the dimensions — in fact, we could always split on the same dimension.

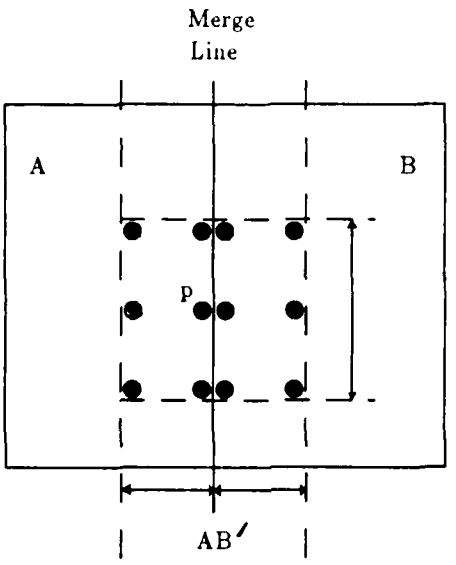


Figure 4: Merging two rectangles to determine closest pair. Only 12 points can fit in the $2\delta_{min} \times 2\delta_{min}$ dashed box such that no two points in either A or B are closer than δ_{min}.

At each merging step, we know the closest pair within each of a pair of merging rectangles A and B and want to find the closest pair in the rectangle $A \cup B$. The closest pair will either be the pair in A, the pair in B, or a pair with one point in A and the other in B. In the last case, the two end points must each lie within $\delta_{min} = min(\delta_A, \delta_B)$ of the boundary between the two rectangles. We will call this region AB'(see Figure 4).

If we look at a point p in AB', no more than 11 other points in AB' can be less than δ_{min} away from p. Figure 4 shows the tightest possible packing. If we have the points in AB' sorted along the merge line, each point can determine the minimum distance to another point in AB' by looking at a fixed number of neighbors in the sorted order (at most 11). Once all points in AB' have found their closest neighbor in AB', we take the minimum of these distances to find $\delta_{AB'}$ and then calculate the desired result: $\delta_{AB} = min(\delta_{min}, \delta_{AB'})$.

We now show how this technique is applied in the scan-model. The merge consists of the following steps (each requires a constant number of calls to the primitives):

1. Derive the vector of points in $A \cup B$ sorted along the direction of the split line. To get this vector, we need only keep the appropriate rank-vector when we construct the k-D tree — remember that when building a k-D tree we had the sorted order for all

dimensions for all rectangles.

2. Determine δ_{min} by taking the *minimum* of δ_A and δ_B. Distribute δ_{min} to all points in the sorted vector of $A \cup B$.

3. Pack elements which are within δ_{min} of the merge line using the **pack** operation into a new sorted vector AB'.

4. Shift this vector to the right and calculate the distance from each point to its neighbor. Repeat this six times to get the six neighbors on each side.

5. Determine $\delta_{AB'}$ by taking the *minimum* distance found in the previous step using a **min-reduce**. Take the minimum of δ_{min} and $\delta_{AB'}$ to get δ_{AB}.

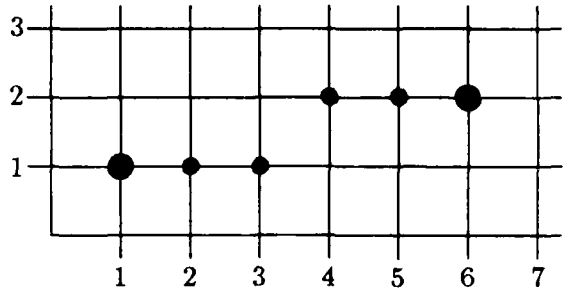
The algorithm will run in $O(\lg n)$ time because the k-D splitting runs in $O(\lg n)$ time and there are $O(\lg n)$ merge steps which, as shown, each step requires constant time.

If the $O(k \lg n)$ space required to store the rank-vectors during the 2-D tree is a *problem*, we can derive the sorted vector for $A \cup B$ on the fly by merging the sorted vectors of A and B. This merge requires $O(\lg \lg n)$ time [10] and will therefore increase the running time of the whole algorithm to $O(\lg n \lg \lg n)$. In the conclusion we mention that it might be reasonable to consider the merge operation as a unit-time primitive of a vector model. If we include a merge primitive, the algorithm will run in $O(\lg n)$ calls to the primitives with $O(n)$ space.

6 Line Drawing

Two dimensional line drawing is the problem of: given a pair of points on a two dimensional grid (the two endpoints of a line), determine what pixels in a finite resolution grid lie on a line between the endpoints. Line drawing is used extensively in practice in generating computer images, especially in computer aided design. In this section we describe a very simple line drawing routine. It generates the same set of pixels as does the simple digital differential analyzer (DDA) serial technique [19]. The routine takes a small constant number of calls to the primitives on vectors at most as long as the number of pixels in the output. Because of the **segment lemma** (Section 2.1), the routine can be used to draw many lines in parallel. The routine we describe has been extended by Salem [25] to render solid objects.

The basic idea of the routine is to calculate the number of pixels in a line and allocate a set of vectors of that length with the line information distributed across the vectors.



$p_1 = (1, 1), \qquad p_2 = (6, 2)$

length = line-length(p_1, p_2) = 5

Δ = increment(p_1, p_2, length) = (1, .2)

pixels = length + 1 = 6

index(pixels)	=	[0	1	2	3	4	5]
distribute(p_1, pixels)	=	[(1, 1)	(1, 1)	(1, 1)	(1, 1)	(1, 1)	(1, 1)]
distribute(Δ, pixels)	=	[(1, .2)	(1, .2)	(1, .2)	(1, .2)	(1, .2)	(1, .2)]
final-position(p_1, Δ, index)	=	[(1, 1)	(2, 1)	(3, 1)	(4, 1)	(5, 1)	(6, 1)]

Figure 5: An example of parallel line drawing.

Then based on the line information and a unique index for each element, the elements can calculate their final position on the grid. Figure 5 illustrates an example.

The code needed to draw a *line* is:

```
define line-length(p1, p2){maximum((p2.x − p1.x), (p2.y − p1.y))}

define increment(p1, p2, length){
    x ← (p2.x − p1.x) / length;
    y ← (p2.y − p1.y) / length}

define final-position(p1, Δ, index){
    x ← p1.x + round(index × Δ.x);
    y ← p1.y + round(index × Δ.y)}

define line-draw(p1, p2){
        length ← line-length(p1, p2);
        Δ ← increment(p1, p2, length);
        pixels ← length + 1;
    final-position(distribute(p1, pixels), distribute(Δ, pixels), index(pixels))}
```

The line-length routine calculates the length of the line. The increment routine calculates the x and y increments between adjacent pixels in the line. The final-position routine calculates the pixel position of a point given one endpoint of the line, the x and y increments of the line, and the position (index) along the line.

The line-draw routine uses the distribute operation to distribute p_1 and the increment (Δ) over (line-length + 1) elements, and uses the index operation to generate a set of consecutive integers for each elements. We need (line-length + 1) elements because we want to include both endpoints.

7 Line of Sight

Given an $\sqrt{n}$ by $\sqrt{n}$ grid of altitudes and an observation point on or above the surface, a *line of sight algorithm* finds all points on the grid visible from the observation point. Figure 6 shows an example. A line of sight algorithm can be applied to help determine where to locate potential eyesores. For example, when designing a building, a highway or a city dump, it is often informative to know from where the "eyesore" will be visible.

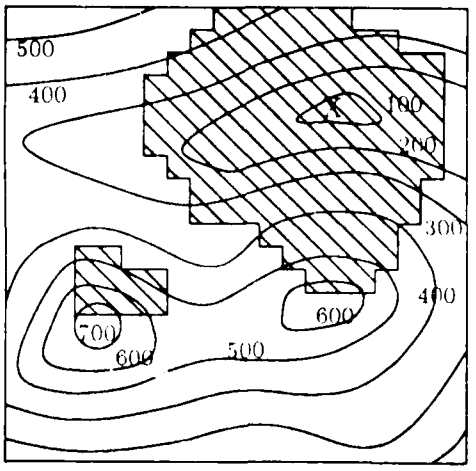


Figure 6: An example of a line of sight problem. The X marks the observation point. The numbers represent the altitude of each contour line. The elements visible from the observation point are shaded.

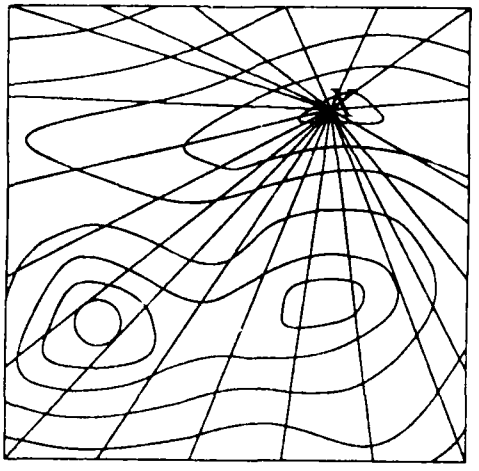


Figure 7: Example of some rays propagating from the observation point.

The algorithm we describe in this section requires $O(1)$ calls to the primitives using vectors of length $O(n)$. The basic idea is to allocate a segment in a vector for every ray that propagates in the plane from the observation point, henceforth referred to as X, to a boundary position (see Figure 7). Based on some calculations on the points in each ray, we can determine if the point is visible.

The algorithm consists of three basic steps.

1. Each point p in the grid calculates the vertical angle between the horizontal plane that passes through X (*the observation point*) and the line from p to X. This is executed by distributing the location of X over all points and calculating the *arctan* of the horizontal difference over the vertical difference.

2. The algorithms allocates a set of rays — one for each boundary grid point — and distributes the angles from each point p in the grid to all the rays it belongs to. Each ray is a segment in a vector we will call the *ray structure*.

3. Following a ray from X to the boundary, a point p is visible if its angle is greater than all the angles that precede it in the ray. This can be determined for all points in all rays with a single segmented **max-scan**, and a comparison.

4. Visibility information is returned back to the grid points. Since a grid point can have a position in many rays, the visibility flags are combined using **or**.

Since steps 1 and 3 should be clear, and step 4 is basically the reverse of step 2, we only describe step 2. To allocate the *ray structure* the algorithm draws a line from the observation point to each boundary element using the routine discussed in Section 6. Each grid point might belong to several of these rays (points near X will belong to more rays than points near the edges). To distribute the angle from a grid point to all the rays it belongs to, the algorithm creates another segmented vector structure — the *copy structure*. In the copy structure the algorithm allocates a segment for each grid point p. The size of the segment for a point p is equal to the number of rays p belongs to — this can be determined from the relative positions of p, X and the boundary. Each point p distributes its angle to its segment in the copy structure using the **distribute** operation.

There is now a 1-to-1 mapping between positions in the copy structure and positions in the ray structure. The algorithm can calculate the permutation indices needed to execute this mapping based on the location of X. Once the angles have been permuted to the ray structure, the algorithm executes step 3. To return the information back to the grid structure after step 3, the algorithm uses the same copy structure but instead of distributing,

it reduces using an or-reduce. At completion, all points visible from any ray are marked and returned.

The longest vectors required by the algorithm will be the vectors of the copy and ray structures. It is not hard to show that for a $\sqrt{n}$ by $\sqrt{n}$ grid, independent of the location of X, these vectors will have length $2n$.

8 Convex Hull

The planar convex hull problem is: given n points in the plane, find which of these points lie on the perimeter of the smallest convex region that contains all points. The planar convex hull problem is probably the most studied problem in computational geometry, both because it is a simple problem, making it easy to study, and because it has many applications – applications range from computer graphics [14] to statistics [17].

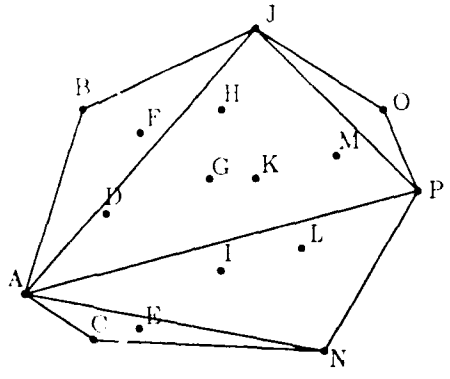
In this section we describe three scan-model based algorithms for determining the convex hull of a set of points. The first two, a parallel Quickhull [22] algorithm and a parallel Jarvis march algorithm [18,2], are very simple and likely to perform well in practice but are not provably optimal. The third algorithms is more complicated and impractical but is theoretically optimal. The algorithm is based on a parallel algorithm designed for the concurrent read exclusive write (CREW) P-RAM model [1,4].

8.1 QuickHull

This is a parallel version of the QuickHull algorithm [22]. The QuickHull algorithm was given its name because of its similarity with the quicksort algorithm. Like quicksort, the quickhull algorithm picks a pivot element, a point; splits the data based on the pivot; and is then recursively applied to each of the split sets. Also like quicksort, the pivot element is not guaranteed to split the data into sets with any particular ratio of sizes, so that in the worst case, the algorithm can require n steps.

Figure 8 shows an example of the quickhull algorithm. The algorithm first splits the points into two sets with a line that passes between the two x extrema — lets call these points l and r. In the scan-model this is executed with a few reduce and distribute operations, some elementwise arithmetic calculations, and a split operation.

The algorithm now recursively splits each of the two subspaces into two using the following steps. It determines for each point p in the subspace the perpendicular distance from the point to the line lr. This can be calculated with a cross product of the lines lr and lp. The algorithm selects the farthest point from the line lr and distributes it to all other



[A B C D E F G H I J K L M N O P]

A [B D F G H J K M O] P [C E I L N]

A [B F] J [O] P N [C E]

A B J O P N C

Figure 8: An example of the QuickHull algorithm. Each vector shows one step of the algorithm. Since A and P are the two x extrema, the line AP is the original split line. J and N are the farthest points in each subspace from AP and are therefore used for the next level of splits. The values outside the brackets are hull points that have already been found.

elements in the subspace — lets call this point t. It should be clear that t lies on the convex hull. Points within the triangle ltr cannot be on the convex hull and are eliminated with a pack operation. The point t is now used to further split each segment based on which of the two sides of the triangle, lt or rt, they fall. The algorithm is now applied to the new segments recursively. The algorithm is completed when all segments are empty.

Each step requires a small constant number of calls to the primitives. As with the serial QuickHull, for m hull points, the algorithm runs in $O(\lg m)$ steps for well distributed hull points, and has a worst case running time of $O(m)$ steps.

8.2 Jarvis March

This is a parallel version of the Jarvis march algorithm. As with the serial version, it will work well when there are only a few points on the hull, such as when the convex hull is a simple polygon. The algorithm starts at an extremum point e and finds the point n that makes the maximum polar angle with e — n is the next point on the hull. The algorithm then finds the maximum polar angle to this point. The step repeats around the hull until we return to the original point. To find each hull point we need a few arithmetic operations and a single max-reduce.

For m hull points, this algorithm requires $O(m)$ steps, but each step is so simple that in some cases the algorithm is faster than the other algorithms mentioned.

8.3 $\sqrt{n}$ Merge Hull

This algorithm is a variation of a parallel algorithm suggested in [1] and independently in [7]. Their algorithm is based on the concurrent read, exclusive write (CREW) P-RAM model. We cannot use their algorithm directly because the scan-model does not permit concurrent access to a single value, a necessary part of their algorithm. The variation we describe keeps all elements that require the same data in a contiguous segment so the data can be distributed using a distribute operation. The contribution of our version is showing how the concurrent read operation can be replaced by the distribute operation and involves a tree search method discussed in the next section. Like the original algorithm, the variation we describe runs with $O(\lg n)$ calls to the primitives. We begin by reviewing the CREW algorithm.

The algorithm sorts the points according to their x coordinate. It slices this ordering into $\sqrt{n}$ equal-sized sets of points and recursively solves the convex hull for each set. It then

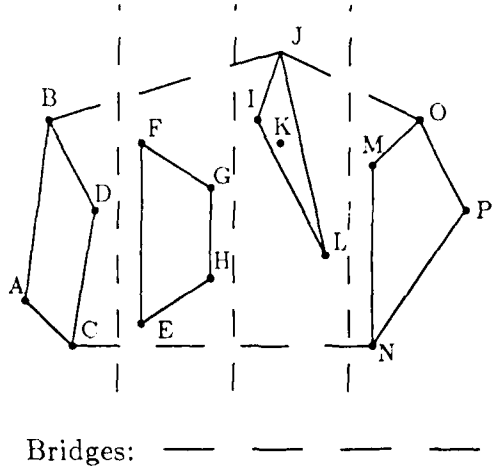


Figure 9: An example of the $\sqrt{n}$ merge hull algorithm. The horizontal dashed lines show the division of the points into $\sqrt{n}$ groups of $\sqrt{n}$ elements each. The subhulls within each group are marked with solid lines. The upper chain is the chain A B J O P.

merges the $\sqrt{n}$ subhulls (see Figure 9). The sort and the merge both take $O(\lg n)$ time[10]. The running time of the algorithm thus has the recurrence relation $T(n) = T(\sqrt{n}) + k \lg n$ which yields $O(\lg n)$ time.

Since the elements can be sorted using existing algorithms we will concentrate on the merging step. The merge is executed in two parts: one finds the upper chain of the convex hull and another finds the lower chain. The upper chain is the section of the convex hull that runs across the top between the two x maxima. In the CREW algorithm the merge of each chain works as follows.

The algorithm assigns an element (a processor) for each pair of subhulls. Since there are $\sqrt{n}$ subhulls, $O(n)$ elements are sufficient. Each of these pairs independently finds the upper tangent line segment[11] between its two subhulls using a serial method of Overmars [21]. This method executes a binary search alternating between the two subhulls, and requires $O(\lg n)$ time. At the k^{th} step of the binary search, an element will either go down the left branch, the right branch or will stay still.

Once the upper tangent lines have been found, the algorithm determines the bridges

[10] The algorithm of Cole [12] can be used for sorting in the CREW model.

[11] An upper tangent line-segment of two sets of points is the line that passes through at least one point from each set so that all other points in the two sets are below the line.

among the $\sqrt{n}$ subhulls. The bridges are the upper tangent line-segments that belong to the upper chain. To find which of the upper tangent lines are bridges, each subhull finds the highest sloped line in both directions (to a point on the right and to a point on the left). If the joint formed by these lines is convex, then both lines are bridges. If the joint formed by the lines is concave, neither are bridges. All edges on a subhull that lie between bridges of that subhull also belong to the convex hull.

This algorithm cannot be implemented directly on the scan-model since each pair of subhulls independently finds the upper tangent-line segments using the algorithm of Overmars, and will therefore require concurrent reads: several pairs, while executing the binary search, will require access to the same elements. To avoid the concurrent read, we place each of the sets of $\sqrt{n}$ points that belong to the same subhull in its own segment. We then use a general binary search method described in the next section to execute the binary search. This search will require $O(\lg n)$ time.

Our variation of the CREW algorithm runs with the same number of calls to the primitives as the original since, as with the original, the sort runs in $O(\lg n)$ time, and, as shown above, the merge also runs in $O(\lg n)$ time. In a sense, this variation trades the the concurrent read capability for the scan capability.

9 Binary Search

In this section we consider the problem of n elements of a set A each executing a binary search on a binary tree T with m vertices. We assume that the tree T is organized in a vector using the standard heap ordering: the root value is stored at $T[1]$ and the two children of a vertex stored at $T[i]$ are stored at $T[2i]$ and $T[2i + 1]$. With a *concurrent-read* primitive, a binary search is simple: each element of A starts by reading the root of T, decides which way to go, and follows a path down to the leaves based on a test and some simple arithmetic at each vertex. Such a search requires concurrent access by many elements of A to a single element of T.

To execute the binary search using the scan primitives instead of a concurrent read primitive we can use a method based on recursive splitting. We start with all the elements of A in a single segment and then split that segment based on whether an element is going to the right or to the left child of the root of T. We then recursively split within each of these segments, based on data from the next level of the tree. Since all the elements of A that are accessing the same vertex of T will be in a contiguous segment, we can use the distribute operation to distribute the value from each vertex of the tree to the elements that

need it.

We now consider a generalization on the simple binary search. At each step, as well as allowing an element of A to go to the left or right child of the vertex of T it is currently at, we allow it to remain at the same vertex of T. This means that there might be elements of A at all levels of the tree instead of at a single level. We also allow new elements to enter the search tree at each step. Figure 10 illustrates how we store the elements of A and an example of a step of the generalized binary search.

To execute a step of the binary search, we must somehow append the elements at a vertex v that remain, with the elements being passed down from the parent of v. To append the elements, we can use the append operation discussed in Section 3. The basic idea is first to separate the elements that remain from those that go to a child into two separate vectors using two pack operations. For the example of Figure 10 this would return:

$$\text{remain} = [a_0]\ []\ [a_4]\ []\ [a_5\ \ a_6]\ []\ [a_7]$$

$$\text{not-remain} = [a_1]\ [a_2]\ [a_3]\ []\ []\ []\ []$$

We then split the ones going to a child based on whether they are going to the left or right child using a split operation. This would return:

$$\text{splitnot-remain} = []\ [a_1]\ [a_2]\ []\ [a_3]\ []\ []\ []\ []\ []\ []\ []\ []\ []$$

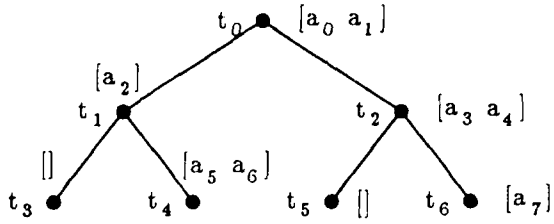
We now shift the segments of the split vector right by one and insert the new elements in the left. Because of the heap order of T, this will cause each segment to go to its child segment. We also truncate the segments that correspond to children of the leaf vertices. These calculations would return:

$$\text{children} = [a_8\ \ a_9]\ []\ [a_1]\ [a_2]\ []\ [a_3]\ []$$

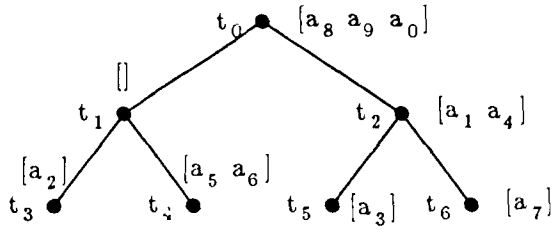
We now append the shifted vector (children) to the vector of elements that remained (remain) using the append operation.

The following routine can be used to execute a step of the binary search. The *remain?* flag specifies elements that stay at the current vertex, and the *right?* flag specifies elements that go to the right branch. *B* are the new elements to be inserted at root.

```
define search-step(A, T, B, remain?, right?){
    remain ← pack(A, remain?);
    not-remain ← pack(A, not(remain?));
    children ← shift-segments-right(B, split(not-remain, right?));
  append(remain, children)}
```



Before



After

$$T = [t_0 \quad t_1 \quad t_2 \quad t_3 \quad t_4 \quad t_5 \quad t_6]$$

$$\begin{aligned} A\text{ (before)} &= [a_0 \quad a_1] \quad [a_2] \quad [a_3 \quad a_4] \quad [] \quad [a_5 \quad a_6] \quad [] \quad [a_7] \\ F &= [\text{ x} \quad \text{r }] \quad [\text{l}] \quad [\text{ l} \quad \text{x }] \quad [] \quad [\text{ x} \quad \text{x }] \quad [] \quad [\text{x}] \end{aligned}$$

$$B = [a_8 \quad a_9]$$

$$A\text{ (after)} = [a_8 \quad a_9 \quad a_0] \quad [] \quad [a_1 \quad a_4] \quad [a_2] \quad [a_5 \quad a_6] \quad [a_3] \quad [a_7]$$

Figure 10: An example of a step of the general binary search technique. We keep a segment in A for each vertex of the tree T such that segment i corresponds to vertex i. Each segment contains all elements at the corresponding vertex. The vector F indicates an example of where each element wants to go during a step of the search (r for right, l for left, and x for remain). The vector B contains new elements entering at the root of the search at that step.

This search step can be used to execute the binary search needed by the $\sqrt{n}$ merge hull algorithm discussed in Section 8.3. For the merge hull, no elements get inserted into the tree, but some elements do remain at their current vertex during a step of the binary search.

Binary search illustrates an important difference between the general programming style used for concurrent read P-RAM models and for vector models. In the P-RAM model, the problem is best thought of as n independent processes each executing its own search on the tree T. In the scan model, we must think of the n elements as a set and break that set into subsets according to which vertex of T each element is accessing. This might just be a philosophical point, but we believe it is important.

10 Conclusions

This paper introduces the idea of a vector model of computation; defines a particular vector model, the scan-model; and describes several algorithms implemented on the scan model. Since many of the algorithms discussed in this paper are variants of known algorithms, we believe that much of the contribution of this paper is to methodology rather than to algorithms. The code we show in this paper with only slight syntactic changes has been used to implement the algorithms described on the Connection Machine.

We believe that the algorithms we describe are very practical for implementation on a wide range of architectures, both serial and parallel, and should in most cases be almost as fast on a particular architecture as algorithm designed specifically for that architecture[12]. This generality is one of the main advantages of the scan-model over the P-RAM models. The advantage arises both because the scan-model is a vector model, allowing efficient implementations on vector processors and single instruction parallel processors, and because it treats the scan operation as taking no more time than a permutation, a realistic assumption for almost all architectures.

In more recent work we have been considering the effect of including other operations as unit time primitives. The operation we have found most promising is a variation of the merge operation[13]. This operation can be implemented efficiently on a wide range of architectures and is useful for many algorithms. To implement the merge operation on serial architectures we can use the standard merge operation, and on parallel architectures we can

[12] This is not true for architectures with low connectivity such as grid architectures or tree architectures.

[13] Given two vectors A and B of numbers, it returns a vector C of length A with indices into the vector B. These indices point to where in B an element in A should merge.

use a variation of Batcher's bitonic merge [7]. Algorithms to construct and manipulate the plane-sweep tree data structure [3,1,5,23] can be greatly simplified with a unit-time merge operation. We have also found the merge primitive useful for manipulating sets. We have also considered sorting as a primitive, but we find it hard to argue that sorting should be assumed to require the same time as a permutation.

We hope that the paper will help spur further interest in designing algorithms for vector models of computation.

Acknowledgments

We would like to thank Charles Leiserson and Guy Steele for their contributions. We would also like to thank Thinking Machines for the opportunity to work on the Connection Machine.

References

[1] Alok Aggarwal, Bernard Chazelle, Leo Guibas, Colm Ó'Dúnlaing, and Chee Yap. Parallel computational geometry. In *Proceedings Symposium on Foundations of Computer Science*, pages 468–477, October 1985.

[2] S. G. Akl. Two remarks on a convex hull algorithm. *Information Processing Letters*, 8:108–109, 1979.

[3] Mikhail J. Atallah, Richard Cole, and Michael T. Goodrich. Cascading divide-and-conquer: a technique for designing parallel algorithms. In *Proceedings Symposium on Foundations of Computer Science*, pages 151–160, October 1987.

[4] Mikhail J. Atallah and Michael T. Goodrich. Efficient parallel solutions to some geometric problems. *Journal of Parallel and Distributed Computing*, 3(4):492–507, December 1986.

[5] Mikhail J. Atallah and Michael T. Goodrich. Efficient plane sweeping in parallel. In *Proceedings ACM Symposium on Theory of Computing*, pages 216–225, 1986.

[6] Kenneth E. Batcher. The flip network of staran. In *Proceedings International Conference on Parallel Processing*, pages 65–71, 1976.

[7] Kenneth E. Batcher. Sorting networks and their applications. In *AFIPS Spring Joint Computer Conference*, pages 307–314, 1968.

[8] Jon L. Bentley. Multidimensional binary search trees used for associative searching. *Communications of the ACM*, 18:509–517, 1975.

[9] Jon L. Bentley and Michael I. Shamos. Divide-and-conquer in multidimensional space. In *Proceedings ACM Symposium on Theory of Computing*, pages 220–230, 1976.

[10] Guy E. Blelloch. *Scans and Other Primitives for Parallel Computation*. PhD thesis, Artificial Intelligence Laboratory, Massachusetts Institute of Technology, Cambridge, MA, to be completed June 1988 (hopefully).

[11] Guy E. Blelloch. Scans as primitive parallel operations. In *Proceedings International Conference on Parallel Processing*, pages 355–362, August 1987.

[12] Richard Cole. Parallel merge sort. In *Proceedings Symposium on Foundations of Computer Science*, pages 511–516, October 1986.

[13] Steven Fortune and James Wyllie. Parallelism in random access machines. In *Proceedings ACM Symposium on Theory of Computing*, pages 114–118, 1978.

[14] H. Freeman. Computer processing of line-drawing images. *Computer Surveys*, 6:57–97, 1974.

[15] Jerome H. Friedman, Jon Louis Bentley, and Raphael Ari Finkel. An algorithm for finding best matches in logarithmic expected time. *ACM Transactions on Mathematical Software*, 3(3):209–226, 1977.

[16] William D. Hillis. *The Connection Machine*. MIT Press, Cambridge, MA, 1985.

[17] J. G. Hocking and G. S. Young. *Topology*. Addison-Wesley, Reading, MA, 1961.

[18] R. A. Jarvis. On the identification of the convex hull on a finite set of points in the plane. *Information Processing Letters*, 2:18–21, 1973.

[19] William M. Newman and Robert F. Sproull. *Principles of Interactive Computer Graphics*. McGraw Hill, New York, 1979.

[20] Stephen M. Omohundro. Efficient algorithms with neural network behavior. *Complex Systems*, 1, 1987.

[21] Mark H. Overmars and Jan Van Leeuwen. Maintenance of configurations in the plane. *Journal of Computer and System Sciences*, 23:166–204, 1981.

[22] Franco P. Preparata and Michael I. Shamos. *Computational Geometry — An Introduction*. Springer-Verlag, New York, 1985.

[23] John H. Reif and Sandeeep Sen. Optimal randomized parallel algorithms for computational geometry. In *Proceedings International Conference on Parallel Processing*, pages 270–277, August 1987.

[24] Richard M. Russell. The CRAY-1 computer system. *Communications of the ACM*, 21(1):63–72, January 1978.

[25] James Salem. **Render: A Rendering Package for the Connection Machine*. Technical Report, Thinking Machines Corporation, November 1987.

[26] Jacob T. Schwartz. Ultracomputers. *ACM Transactions on Programming Languages and Systems*, 2(4):484–521, October 1980.

END

DATE

FILMED

6-1988

DTIC